Laravel AI SDK

# Introduction

The Laravel AI SDK provides a unified, expressive API for interacting with AI providers such as OpenAI, Anthropic, Gemini, and more. With the AI SDK, you can build intelligent agents with tools and structured output, generate images, synthesize and transcribe audio, create vector embeddings, and much more — all using a consistent, Laravel-friendly interface.

# Installation

You can install the Laravel AI SDK via Composer:

```
composer require laravel/ai
```

Next, you should publish the AI SDK configuration and migration files using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --provider="Laravel\Ai\AiServiceProvider"
```

Finally, you should run your application's database migrations. This will create a `agent_conversations` and `agent_conversation_messages` table that the AI SDK uses to power its conversation storage:

```
php artisan migrate
```

Configuration

You may define your AI provider credentials in your application's `config/ai.php` configuration file or as environment variables in your application's `.env` file:

```
ANTHROPIC_API_KEY=
COHERE_API_KEY=
ELEVENLABS_API_KEY=
GEMINI_API_KEY=
MISTRAL_API_KEY=
OLLAMA_API_KEY=
OPENAI_API_KEY=
JINA_API_KEY=
VOYAGEAI_API_KEY=
XAI_API_KEY=
```

The default models used for text, images, audio, transcription, and embeddings may also be configured in your application's `config/ai.php` configuration file.

Custom Base URLs

By default, the Laravel AI SDK connects directly to each provider's public API endpoint. However, you may need to route requests through a different endpoint - for example, when using a proxy service to centralize API key management, implement rate limiting, or route traffic through a corporate gateway.

You may configure custom base URLs by adding a `url` parameter to your provider configuration:

```
'providers' => [
    'openai' => [
        'driver' => 'openai',
        'key' => env('OPENAI_API_KEY'),
        'url' => env('OPENAI_BASE_URL'),
    ],

    'anthropic' => [
        'driver' => 'anthropic',
        'key' => env('ANTHROPIC_API_KEY'),
        'url' => env('ANTHROPIC_BASE_URL'),
    ],
],
```

This is useful when routing requests through a proxy service (such as LiteLLM or Azure OpenAI Gateway) or using alternative endpoints.

Custom base URLs are supported for the following providers: OpenAI, Anthropic, Gemini, Groq, Cohere, DeepSeek, xAI, and OpenRouter.

Provider Support

The AI SDK supports a variety of providers across its features. The following table summarizes which providers are available for each feature:

Feature	Providers
Text	OpenAI, Anthropic, Gemini, Azure, Groq, xAI, DeepSeek, Mistral, Ollama
Images	OpenAI, Gemini, xAI
TTS	OpenAI, ElevenLabs
STT	OpenAI, ElevenLabs, Mistral
Embeddings	OpenAI, Gemini, Azure, Cohere, Mistral, Jina, VoyageAI
Reranking	Cohere, Jina
Files	OpenAI, Anthropic, Gemini

The `Laravel\Ai\Enums\Lab` enum may be used to reference providers throughout your code instead of using plain strings:

```php
use Laravel\Ai\Enums\Lab;

Lab::Anthropic;
Lab::OpenAI;
Lab::Gemini;
// ...
```

Agents

Agents are the fundamental building block for interacting with AI providers in the Laravel AI SDK. Each agent is a dedicated PHP class that encapsulates the instructions, conversation context, tools, and output schema needed to interact with a large language model. Think of an agent as a specialized assistant — a sales coach, a document analyzer, a support bot — that you configure once and prompt as needed throughout your application.

You can create an agent via the `make:agent` Artisan command:

```
php artisan make:agent SalesCoach

php artisan make:agent SalesCoach --structured
```

Within the generated agent class, you can define the system prompt / instructions, message context, available tools, and output schema (if applicable):

```
<?php

namespace App\Ai\Agents;

use App\Ai\Tools\RetrievePreviousTranscripts;
use App\Models\History;
use App\Models\User;
use Illuminate\Contracts\JsonSchema\JsonSchema;
use Laravel\Ai\Contracts\Agent;
use Laravel\Ai\Contracts\Conversational;
use Laravel\Ai\Contracts\HasStructuredOutput;
use Laravel\Ai\Contracts\HasTools;
use Laravel\Ai\Messages\Message;
use Laravel\Ai\Promptable;
use Stringable;

class SalesCoach implements Agent, Conversational, HasTools, HasStructuredOu
{
    use Promptable;

    public function __construct(public User $user) {}

    /**
     * Get the instructions that the agent should follow.
     */
    public function instructions(): Stringable|string
    {
        return 'You are a sales coach, analyzing transcripts and providing f
    }

    /**
     * Get the list of messages comprising the conversation so far.
     */
    public function messages(): iterable
    {
        return History::where('user_id', $this->user->id)
            ->latest()
            ->limit(50)
            ->get()
```

```php
            ->reverse()
            ->map(function ($message) {
                return new Message($message->role, $message->content);
            })->all();
    }

    /**
     * Get the tools available to the agent.
     *
     * @return Tool[]
     */
    public function tools(): iterable
```

```php
        ];
    }

    /**
     * Get the agent's structured output schema definition.
     */
    public function schema(JsonSchema $schema): array
    {
        return [
            'feedback' => $schema->string()->required(),
            'score' => $schema->integer()->min(1)->max(10)->required(),
        ];
    }
}
```

# Prompting

To prompt an agent, first create an instance using the `make` method or standard instantiation, then call `prompt`:

```php
$response = (new SalesCoach)
    ->prompt('Analyze this sales transcript...');

$response = SalesCoach::make()
    ->prompt('Analyze this sales transcript...');

return (string) $response;
```

The `make` method resolves your agent from the container, allowing automatic dependency injection. You may also pass arguments to the agent's constructor:

```php
$agent = SalesCoach::make(user: $user);
```

By passing additional arguments to the `prompt` method, you may override the default provider, model, or HTTP timeout when prompting:

```php
$response = (new SalesCoach)->prompt(
    'Analyze this sales transcript...',
    provider: Lab::Anthropic,
    model: 'claude-haiku-4-5-20251001',
    timeout: 120,
);
```

# Conversation Context

If your agent implements the `Conversational` interface, you may use the `messages` method to return the previous conversation context, if applicable:

```php
use App\Models\History;
use Laravel\Ai\Messages\Message;

/**
 * Get the list of messages comprising the conversation so far.
 */
public function messages(): iterable
{
    return History::where('user_id', $this->user->id)
        ->latest()
        ->limit(50)
        ->get()
        ->reverse()
        ->map(function ($message) {
            return new Message($message->role, $message->content);
        })->all();
}
```

# Remembering Conversations

Before using the `RemembersConversations` trait, you should publish and run the AI SDK migrations using the `vendor:publish` Artisan command. These migrations will create the necessary database tables to store conversations.

If you would like Laravel to automatically store and retrieve conversation history for your agent, you may use the `RemembersConversations` trait. This trait provides a simple way to persist conversation messages to the database without manually implementing the `Conversational` interface:

```php
<?php

namespace App\Ai\Agents;

use Laravel\Ai\Concerns\RemembersConversations;
use Laravel\Ai\Contracts\Agent;
use Laravel\Ai\Contracts\Conversational;
use Laravel\Ai\Promptable;

class SalesCoach implements Agent, Conversational
{
    use Promptable, RemembersConversations;

    /**
     * Get the instructions that the agent should follow.
     */
    public function instructions(): string
    {
        return 'You are a sales coach...';
    }
}
```

To start a new conversation for a user, call the `forUser` method before prompting:

```php
$response = (new SalesCoach)->forUser($user)->prompt('Hello!');

$conversationId = $response->conversationId;
```

The conversation ID is returned on the response and can be stored for future reference, or you can retrieve all of a user's conversations from the `agent_conversations` table directly.

To continue an existing conversation, use the `continue` method:

```php
$response = (new SalesCoach)
    ->continue($conversationId, as: $user)
    ->prompt('Tell me more about that.');
```

When using the `RemembersConversations` trait, previous messages are automatically loaded and included in the conversation context when prompting. New messages (both user and assistant) are automatically stored after each interaction.

# Structured Output

If you would like your agent to return structured output, implement the `HasStructuredOutput` interface, which requires that your agent define a `schema` method:

```php
<?php

namespace App\Ai\Agents;

use Illuminate\Contracts\JsonSchema\JsonSchema;
use Laravel\Ai\Contracts\Agent;
use Laravel\Ai\Contracts\HasStructuredOutput;
use Laravel\Ai\Promptable;

class SalesCoach implements Agent, HasStructuredOutput
{
    use Promptable;

    // ...

    /**
     * Get the agent's structured output schema definition.
     */
    public function schema(JsonSchema $schema): array
    {
        return [
            'score' => $schema->integer()->required(),
        ];
    }
}
```

When prompting an agent that returns structured output, you can access the returned `StructuredAgentResponse` like an array:

```php
$response = (new SalesCoach)->prompt('Analyze this sales transcript...');

return $response['score'];
```

# Attachments

When prompting, you may also pass attachments with the prompt to allow the model to inspect images and documents:

```php
use App\Ai\Agents\SalesCoach;
use Laravel\Ai\Files;

$response = (new SalesCoach)->prompt(
    'Analyze the attached sales transcript...',
    attachments: [
        Files\Document::fromStorage('transcript.pdf') // Attach a document f
        Files\Document::fromPath('/home/laravel/transcript.md') // Attach a 
        $request->file('transcript'), // Attach an uploaded file...
    ]
);
```

Likewise, the `Laravel\Ai\Files\Image` class may be used to attach images to a prompt:

```php
use App\Ai\Agents\ImageAnalyzer;
use Laravel\Ai\Files;

$response = (new ImageAnalyzer)->prompt(
    'What is in this image?',
    attachments: [
        Files\Image::fromStorage('photo.jpg') // Attach an image from a file
        Files\Image::fromPath('/home/laravel/photo.jpg') // Attach an image 
        $request->file('photo'), // Attach an uploaded file...
    ]
);
```

# Streaming

You may stream an agent's response by invoking the `stream` method. The returned `StreamableAgentResponse` may be returned from a route to automatically send a streaming response (SSE) to the client:

```php
use App\Ai\Agents\SalesCoach;

Route::get('/coach', function () {
    return (new SalesCoach)->stream('Analyze this sales transcript...');
});
```

The `then` method may be used to provide a closure that will be invoked when the entire response has been streamed to the client:

```php
use App\Ai\Agents\SalesCoach;
use Laravel\Ai\Responses\StreamedAgentResponse;

Route::get('/coach', function () {
    return (new SalesCoach)
        ->stream('Analyze this sales transcript...')
        ->then(function (StreamedAgentResponse $response) {
            // $response->text, $response->events, $response->usage...
        });
});
```

Alternatively, you may iterate through the streamed events manually:

```php
$stream = (new SalesCoach)->stream('Analyze this sales transcript...');

foreach ($stream as $event) {
    // ...
}
```

Streaming Using the Vercel AI SDK Protocol

You may stream the events using the Vercel AI SDK stream protocol by invoking the `usingVercelDataProtocol` method on the streamable response:

```php
use App\Ai\Agents\SalesCoach;

Route::get('/coach', function () {
    return (new SalesCoach)
        ->stream('Analyze this sales transcript...')
        ->usingVercelDataProtocol();
});
```

Broadcasting

You may broadcast streamed events in a few different ways. First, you can simply invoke the `broadcast` or `broadcastNow` method on a streamed event:

```php
use App\Ai\Agents\SalesCoach;
use Illuminate\Broadcasting\Channel;
```

```

$stream = (new SalesCoach)->stream('Analyze this sales transcript...');

foreach ($stream as $event) {
    $event->broadcast(new Channel('channel-name'));
}
```

Or, you can invoke an agent's `broadcastOnQueue` method to queue the agent operation and broadcast the streamed events as they are available:

```
(new SalesCoach)->broadcastOnQueue(
    'Analyze this sales transcript...'
    new Channel('channel-name'),
);
```

# Queueing

Using an agent's `queue` method, you may prompt the agent, but allow it to process the response in the background, keeping your application feeling fast and responsive. The `then` and `catch` methods may be used to register closures that will be invoked when a response is available or if an exception occurs:

```
use Illuminate\Http\Request;
use Laravel\Ai\Responses\AgentResponse;
use Throwable;

Route::post('/coach', function (Request $request) {
    return (new SalesCoach)
        ->queue($request->input('transcript'))
        ->then(function (AgentResponse $response) {
            // ...
        })
        ->catch(function (Throwable $e) {
            // ...
        });

    return back();
});
```

# Tools

Tools may be used to give agents additional functionality that they can utilize while responding to prompts. Tools can be created using the `make:tool` Artisan command:

```
php artisan make:tool RandomNumberGenerator
```

The generated tool will be placed in your application's `app/Ai/Tools` directory. Each tool contains a `handle` method that will be invoked by the agent when it needs to utilize the tool:

```php
<?php

namespace App\Ai\Tools;

use Illuminate\Contracts\JsonSchema\JsonSchema;
use Laravel\Ai\Contracts\Tool;
use Laravel\Ai\Tools\Request;
use Stringable;

class RandomNumberGenerator implements Tool
{
    /**
     * Get the description of the tool's purpose.
     */
    public function description(): Stringable|string
    {
        return 'This tool may be used to generate cryptographically secure r
    }

    /**
     * Execute the tool.
     */
    public function handle(Request $request): Stringable|string
    {
        return (string) random_int($request['min'], $request['max']);
    }

    /**
     * Get the tool's schema definition.
     */
    public function schema(JsonSchema $schema): array
    {
        return [
            'min' => $schema->integer()->min(0)->required(),
            'max' => $schema->integer()->required(),
        ];
    }
}
```

Once you have defined your tool, you may return it from the `tools` method of any of your agents:

```php
use App\Ai\Tools\RandomNumberGenerator;

/**
 * Get the tools available to the agent.
 *
 * @return Tool[]
 */
public function tools(): iterable
{
    return [
        new RandomNumberGenerator,
    ];
}
```

# Similarity Search

The `SimilaritySearch` tool allows agents to search for documents similar to a given query using vector embeddings stored in your database. This is useful for retrieval-augmented generation (RAG) when you want to give agents access to search your application's data.

The simplest way to create a similarity search tool is using the `usingModel` method with an Eloquent model that has vector embeddings:

```php
use App\Models\Document;
use Laravel\Ai\Tools\SimilaritySearch;

public function tools(): iterable
{
    return [
        SimilaritySearch::usingModel(Document::class, 'embedding'),
    ];
}
```

The first argument is the Eloquent model class, and the second argument is the column containing the vector embeddings.

You may also provide a minimum similarity threshold between `0.0` and `1.0` and a closure to customize the query:

```php
SimilaritySearch::usingModel(
    model: Document::class,
        column: 'embedding',
        minSimilarity: 0.7,
        limit: 10,
        query: fn ($query) => $query->where('published', true),
    ),
```

For more control, you may create a similarity search tool with a custom closure that returns the search results:

```php
use App\Models\Document;
use Laravel\Ai\Tools\SimilaritySearch;

public function tools(): iterable
{
    return [
        new SimilaritySearch(using: function (string $query) {
            return Document::query()
                ->where('user_id', $this->user->id)
                ->whereVectorSimilarTo('embedding', $query)
                ->limit(10)
                ->get();
        }),
    ];
}
```

You may customize the tool's description using the `withDescription` method:

```php
SimilaritySearch::usingModel(Document::class, 'embedding')
    ->withDescription('Search the knowledge base for relevant articles.'),
```

# Provider Tools

Provider tools are special tools implemented natively by AI providers, offering capabilities like web searching, URL fetching, and file searching. Unlike regular tools, provider tools are executed by the provider itself rather than your application.

Provider tools can be returned by your agent's `tools` method.

# Web Search

The `WebSearch` provider tool allows agents to search the web for real-time information. This is useful for answering questions about current events, recent data, or topics that may have changed since the model's training cutoff.

Supported Providers: Anthropic, OpenAI, Gemini

```php
use Laravel\Ai\Providers\Tools\WebSearch;

public function tools(): iterable
{
    return [
        new WebSearch,
    ];
}
```

You may configure the web search tool to limit the number of searches or restrict results to specific domains:

```php
(new WebSearch)->max(5)->allow(['laravel.com', 'php.net']),
```

To refine search results based on user location, use the `location` method:

```php
(new WebSearch)->location(
    city: 'New York',
    region: 'NY',
    country: 'US'
);
```

# Web Fetch

The `WebFetch` provider tool allows agents to fetch and read the contents of web pages. This is useful when you need the agent to analyze specific URLs or retrieve detailed information from known web pages.

Supported providers: Anthropic, Gemini

```php
use Laravel\Ai\Providers\Tools\WebFetch;

public function tools(): iterable
{
    return [
        new WebFetch,
    ];
}
```

You may configure the web fetch tool to limit the number of fetches or restrict to specific domains:

```php
(new WebFetch)->max(3)->allow(['docs.laravel.com']),
```

File Search

The `FileSearch` provider tool allows agents to search through files stored in vector stores. This enables retrieval-augmented generation (RAG) by allowing the agent to search your uploaded documents for relevant information.

Supported providers: OpenAI, Gemini

```php
use Laravel\Ai\Providers\Tools\FileSearch;

public function tools(): iterable
{
    return [
        new FileSearch(stores: ['store_id']),
    ];
}
```

You may provide multiple vector store IDs to search across multiple stores:

```php
new FileSearch(stores: ['store_1', 'store_2']);
```

If your files have metadata, you may filter the search results by providing a `where` argument. For simple equality filters, pass an array:

```php
new FileSearch(stores: ['store_id'], where: [
    'author' => 'Taylor Otwell',
    'year' => 2026,
]);
```

For more complex filters, you may pass a closure that receives a `FileSearchQuery` instance:

```php
use Laravel\Ai\Providers\Tools\FileSearchQuery;

new FileSearch(stores: ['store_id'], where: fn (FileSearchQuery $query) =>
    $query->where('author', 'Taylor Otwell')
```

```php
            ->whereNot('status', 'draft')
            ->whereIn('category', ['news', 'updates'])
    );
```

Middleware

Agents support middleware, allowing you to intercept and modify prompts before they are sent to the provider. To add middleware to an agent, implement the `HasMiddleware` interface and define a `middleware` method that returns an array of middleware classes:

```php
<?php

namespace App\Ai\Agents;

use Laravel\Ai\Contracts\Agent;
use Laravel\Ai\Contracts\HasMiddleware;
use Laravel\Ai\Promptable;

class SalesCoach implements Agent, HasMiddleware
{
    use Promptable;

    // ...

    /**
     * Get the agent's middleware.
     */
    public function middleware(): array
    {
        return [
            new LogPrompts,
        ];
    }
}
```

Each middleware class should define a `handle` method that receives the `AgentPrompt` and a `Closure` to pass the prompt to the next middleware:

```php
<?php

namespace App\Ai\Middleware;

use Closure;
use Laravel\Ai\Prompts\AgentPrompt;

class LogPrompts
```

```php
{
    /**
     * Handle the incoming prompt.
     */
    public function handle(AgentPrompt $prompt, Closure $next)
    {
        Log::info('Prompting agent', ['prompt' => $prompt->prompt]);

        return $next($prompt);
    }
}
```

You may use the `then` method on the response to execute code after the agent has finished processing. This works for both synchronous and streaming responses:

```php
public function handle(AgentPrompt $prompt, Closure $next)
{
    return $next($prompt)->then(function (AgentResponse $response) {
        Log::info('Agent responded', ['text' => $response->text]);
    });
}
```

# Anonymous Agents

Sometimes you may want to quickly interact with a model without creating a dedicated agent class. You can create an ad-hoc, anonymous agent using the `agent` function:

```php
use function Laravel\Ai\{agent};

$response = agent(
    instructions: 'You are an expert at software development.',
    messages: [],
    tools: [],
)->prompt('Tell me about Laravel')
```

Anonymous agents may also produce structured output:

```php
use Illuminate\Contracts\JsonSchema\JsonSchema;

use function Laravel\Ai\{agent};

$response = agent(
    schema: fn (JsonSchema $schema) => [
        'number' => $schema->integer()->required(),
    ],
)->prompt('Generate a random number less than 100')
```

Agent Configuration

You may configure text generation options for an agent using PHP attributes. The following attributes are available:

- `MaxSteps`: The maximum number of steps the agent may take when using tools.
- `MaxTokens`: The maximum number of tokens the model may generate.
- `Model`: The model the agent should use.
- `Provider`: The AI provider (or providers for failover) to use for the agent.
- `Temperature`: The sampling temperature to use for generation (0.0 to 1.0).
- `Timeout`: The HTTP timeout in seconds for agent requests (default: 60).
- `UseCheapestModel`: Use the provider's cheapest text model for cost optimization.
- `UseSmartestModel`: Use the provider's most capable text model for complex tasks.

```php
<?php

namespace App\Ai\Agents;

use Laravel\Ai\Attributes\MaxSteps;
use Laravel\Ai\Attributes\MaxTokens;
use Laravel\Ai\Attributes\Model;
use Laravel\Ai\Attributes\Provider;
use Laravel\Ai\Attributes\Temperature;
use Laravel\Ai\Attributes\Timeout;
use Laravel\Ai\Contracts\Agent;
use Laravel\Ai\Enums\Lab;
use Laravel\Ai\Promptable;

#[Provider(Lab::Anthropic)]
#[Model('claude-haiku-4-5-20251001')]
#[MaxSteps(10)]
#[MaxTokens(4096)]
#[Temperature(0.7)]
#[Timeout(120)]
class SalesCoach implements Agent
{
    use Promptable;

    // ...
}
```

The `UseCheapestModel` and `UseSmartestModel` attributes allow you to automatically select the most cost-effective or most capable model for a given provider without specifying a model name. This is useful when you want to optimize for cost or capability across different providers:

```php
use Laravel\Ai\Attributes\UseCheapestModel;
use Laravel\Ai\Attributes\UseSmartestModel;
use Laravel\Ai\Contracts\Agent;
use Laravel\Ai\Promptable;

#[UseCheapestModel]
class SimpleSummarizer implements Agent
{
    use Promptable;

    // Will use the cheapest model (e.g., Haiku)...
}

#[UseSmartestModel]
class ComplexReasoner implements Agent
{
    use Promptable;

    // Will use the most capable model (e.g., Opus)...
}
```

# Images

The `Laravel\Ai\Image` class may be used to generate images using the `openai`, `gemini`, or `xai` providers:

```php
use Laravel\Ai\Image;

$image = Image::of('A donut sitting on the kitchen counter')->generate();

$rawContent = (string) $image;
```

The `square`, `portrait`, and `landscape` methods may be used to control the aspect ratio of the image, while the `quality` method may be used to guide the model on final image quality (`high`, `medium`, `low`). The `timeout` method may be used to specify the HTTP timeout in seconds:

```php
use Laravel\Ai\Image;

$image = Image::of('A donut sitting on the kitchen counter')
    ->quality('high')
    ->landscape()
    ->timeout(120)
    ->generate();
```

You may attach reference images using the `attachments` method:

```php
use Laravel\Ai\Files;
use Laravel\Ai\Image;

$image = Image::of('Update this photo of me to be in the style of an impress
    ->attachments([
        Files\Image::fromStorage('photo.jpg'),
        // Files\Image::fromPath('/home/laravel/photo.jpg'),
        // Files\Image::fromUrl('https://example.com/photo.jpg'),
        // $request->file('photo'),
    ])
    ->landscape()
    ->generate();
```

Generated images may be easily stored on the default disk configured in your application's `config/filesystems.php` configuration file:

```php
$image = Image::of('A donut sitting on the kitchen counter');

$path = $image->store();
$path = $image->storeAs('image.jpg');
$path = $image->storePublicly();
$path = $image->storePubliclyAs('image.jpg');
```

Image generation may also be queued:

```php
use Laravel\Ai\Image;
use Laravel\Ai\Responses\ImageResponse;

Image::of('A donut sitting on the kitchen counter')
    ->portrait()
    ->queue()
    ->then(function (ImageResponse $image) {
        $path = $image->store();

        // ...
    });
```

# Audio

The `Laravel\Ai\Audio` class may be used to generate audio from the given text:

```
use Laravel\Ai\Audio;

$audio = Audio::of('I love coding with Laravel.')->generate();

$rawContent = (string) $audio;
```

The `male`, `female`, and `voice` methods may be used to determine the voice of the generated audio:

```
$audio = Audio::of('I love coding with Laravel.')
    ->female()
    ->generate();

$audio = Audio::of('I love coding with Laravel.')
    ->voice('voice-id-or-name')
    ->generate();
```

Similarly, the `instructions` method may be used to dynamically coach the model on how the generated audio should sound:

```
$audio = Audio::of('I love coding with Laravel.')
    ->female()
    ->instructions('Said like a pirate')
    ->generate();
```

Generated audio may be easily stored on the default disk configured in your application's `config/filesystems.php` configuration file:

```
$audio = Audio::of('I love coding with Laravel.')->generate();

$path = $audio->store();
```

```
$path = $audio->storeAs('audio.mp3');
$path = $audio->storePublicly();
$path = $audio->storePubliclyAs('audio.mp3');
```

Audio generation may also be queued:

```
use Laravel\Ai\Audio;
use Laravel\Ai\Responses\AudioResponse;

Audio::of('I love coding with Laravel.')
    ->queue()
    ->then(function (AudioResponse $audio) {
        $path = $audio->store();

        // ...
    });
```

# Transcriptions

The `Laravel\Ai\Transcription` class may be used to generate a transcript of the given audio:

```
use Laravel\Ai\Transcription;

$transcript = Transcription::fromPath('/home/laravel/audio.mp3')->generate();
$transcript = Transcription::fromStorage('audio.mp3')->generate();
$transcript = Transcription::fromUpload($request->file('audio'))->generate();

return (string) $transcript;
```

The `diarize` method may be used to indicate you would like the response to include the diarized transcript in addition to the raw text transcript, allowing you to access the segmented transcript by speaker:

```
$transcript = Transcription::fromStorage('audio.mp3')
    ->diarize()
    ->generate();
```

Transcription generation may also be queued:

```php
use Laravel\Ai\Transcription;
use Laravel\Ai\Responses\TranscriptionResponse;

Transcription::fromStorage('audio.mp3')
    ->queue()
    ->then(function (TranscriptionResponse $transcript) {
        // ...
    });
```

# Embeddings

You may easily generate vector embeddings for any given string using the new `toEmbeddings` method available via Laravel's `Stringable` class:

```php
use Illuminate\Support\Str;

$embeddings = Str::of('Napa Valley has great wine.')->toEmbeddings();
```

Alternatively, you may use the `Embeddings` class to generate embeddings for multiple inputs at once:

```php
use Laravel\Ai\Embeddings;

$response = Embeddings::for([
    'Napa Valley has great wine.',
    'Laravel is a PHP framework.',
])->generate();

$response->embeddings; // [[0.123, 0.456, ...], [0.789, 0.012, ...]]
```

You may specify the dimensions and provider for the embeddings:

```php
$response = Embeddings::for(['Napa Valley has great wine.'])
    ->dimensions(1536)
    ->generate(Lab::OpenAI, 'text-embedding-3-small');
```

# Querying Embeddings

Once you have generated embeddings, you will typically store them in a `vector` column in your database for later querying. Laravel provides native support for vector columns on

PostgreSQL via the `pgvector` extension. To get started, define a `vector` column in your migration, specifying the number of dimensions:

```php
Schema::ensureVectorExtensionExists();

Schema::create('documents', function (Blueprint $table) {
    $table->id();
    $table->string('title');
    $table->text('content');
    $table->vector('embedding', dimensions: 1536);
    $table->timestamps();
});
```

You may also add a vector index to speed up similarity searches. When calling `index` on a vector column, Laravel will automatically create an HNSW index with cosine distance:

```php
$table->vector('embedding', dimensions: 1536)->index();
```

On your Eloquent model, you should cast the vector column to an `array`:

```php
protected function casts(): array
{
    return [
        'embedding' => 'array',
    ];
}
```

To query for similar records, use the `whereVectorSimilarTo` method. This method filters results by a minimum cosine similarity (between `0.0` and `1.0`, where `1.0` is identical) and orders the results by similarity:

```php
use App\Models\Document;

$documents = Document::query()
    ->whereVectorSimilarTo('embedding', $queryEmbedding, minSimilarity: 0.4)
    ->limit(10)
    ->get();
```

The `$queryEmbedding` may be an array of floats or a plain string. When a string is given, Laravel will automatically generate embeddings for it:

```php
$documents = Document::query()
    ->whereVectorSimilarTo('embedding', 'best wineries in Napa Valley')
    ->limit(10)
    ->get();
```

If you need more control, you may use the lower-level `whereVectorDistanceLessThan`, `selectVectorDistance`, and `orderByVectorDistance` methods independently:

```php
$documents = Document::query()
    ->select('*')
    ->selectVectorDistance('embedding', $queryEmbedding, as: 'distance')
    ->whereVectorDistanceLessThan('embedding', $queryEmbedding, maxDistance:
    ->orderByVectorDistance('embedding', $queryEmbedding)
    ->limit(10)
    ->get();
```

If you would like to give an agent the ability to perform similarity searches as a tool, check out the Similarity Search tool documentation.

Vector queries are currently only supported on PostgreSQL connections using the `pgvector` extension.

# Caching Embeddings

Embedding generation can be cached to avoid redundant API calls for identical inputs. To enable caching, set the `ai.caching.embeddings.cache` configuration option to `true`:

```php
'caching' => [
    'embeddings' => [
        'cache' => true,
        'store' => env('CACHE_STORE', 'database'),
        // ...
    ],
],
```

When caching is enabled, embeddings are cached for 30 days. The cache key is based on the provider, model, dimensions, and input content, ensuring that identical requests return cached results while different configurations generate fresh embeddings.

You may also enable caching for a specific request using the `cache` method, even when global caching is disabled:

```php
$response = Embeddings::for(['Napa Valley has great wine.'])
    ->cache()
    ->generate();
```

You may specify a custom cache duration in seconds:

```php
$response = Embeddings::for(['Napa Valley has great wine.'])
    ->cache(seconds: 3600) // Cache for 1 hour
    ->generate();
```

The `toEmbeddings` Stringable method also accepts a `cache` argument:

```php
// Cache with default duration...
$embeddings = Str::of('Napa Valley has great wine.')->toEmbeddings(cache: tru

// Cache for a specific duration...
$embeddings = Str::of('Napa Valley has great wine.')->toEmbeddings(cache: 360
```

# Reranking

Reranking allows you to reorder a list of documents based on their relevance to a given query. This is useful for improving search results by using semantic understanding:

The `Laravel\Ai\Reranking` class may be used to rerank documents:

```php
use Laravel\Ai\Reranking;

$response = Reranking::of([
    'Django is a Python web framework.',
    'Laravel is a PHP web application framework.',
    'React is a JavaScript library for building user interfaces.',
])->rerank('PHP frameworks');

// Access the top result...
$response->first()->document; // "Laravel is a PHP web application framework
$response->first()->score;    // 0.95
$response->first()->index;    // 1 (original position)
```

The `limit` method may be used to restrict the number of results returned:

```php
$response = Reranking::of($documents)
    ->limit(5)
    ->rerank('search query');
```

Reranking Collections

For convenience, Laravel collections may be reranked using the `rerank` macro. The first argument specifies which field(s) to use for reranking, and the second argument is the query:

```php
// Rerank by a single field...
$posts = Post::all()
    ->rerank('body', 'Laravel tutorials');

// Rerank by multiple fields (sent as JSON)...
$reranked = $posts->rerank(['title', 'body'], 'Laravel tutorials');

// Rerank using a closure to build the document...
$reranked = $posts->rerank(
    fn ($post) => $post->title.': '.$post->body,
    'Laravel tutorials'
);
```

You may also limit the number of results and specify a provider:

```php
$reranked = $posts->rerank(
    by: 'content',
    query: 'Laravel tutorials',
    limit: 10,
    provider: Lab::Cohere
);
```

Files

The `Laravel\Ai\Files` class or the individual file classes may be used to store files with your AI provider for later use in conversations. This is useful for large documents or files you want to reference multiple times without re-uploading:

```php
use Laravel\Ai\Files\Document;
use Laravel\Ai\Files\Image;

// Store a file from a local path...
$response = Document::fromPath('/home/laravel/document.pdf')->put();
$response = Image::fromPath('/home/laravel/photo.jpg')->put();

// Store a file that is stored on a filesystem disk...
$response = Document::fromStorage('document.pdf', disk: 'local')->put();
$response = Image::fromStorage('photo.jpg', disk: 'local')->put();

// Store a file that is stored on a remote URL...
$response = Document::fromUrl('https://example.com/document.pdf')->put();
$response = Image::fromUrl('https://example.com/photo.jpg')->put();

return $response->id;
```

You may also store raw content or uploaded files:

```php
use Laravel\Ai\Files;
use Laravel\Ai\Files\Document;

// Store raw content...
$stored = Document::fromString('Hello, World!', 'text/plain')->put();

// Store an uploaded file...
$stored = Document::fromUpload($request->file('document'))->put();
```

Once a file has been stored, you may reference the file when generating text via agents instead of re-uploading the file:

```php
use App\Ai\Agents\SalesCoach;
use Laravel\Ai\Files;

$response = (new SalesCoach)->prompt(
    'Analyze the attached sales transcript...'
    attachments: [
        Files\Document::fromId('file-id') // Attach a stored document...
    ]
);
```

To retrieve a previously stored file, use the `get` method on a file instance:

```php
use Laravel\Ai\Files\Document;

$file = Document::fromId('file-id')->get();

$file->id;
$file->mimeType();
```

To delete a file from the provider, use the `delete` method:

```php
Document::fromId('file-id')->delete();
```

By default, the `Files` class uses the default AI provider configured in your application's `config/ai.php` configuration file. For most operations, you may specify a different provider using the `provider` argument:

```php
$response = Document::fromPath(
    '/home/laravel/document.pdf'
)->put(provider: Lab::Anthropic);
```

# Using Stored Files in Conversations

Once a file has been stored with a provider, you may reference it in agent conversations using the `fromId` method on the `Document` or `Image` classes:

```php
use App\Ai\Agents\DocumentAnalyzer;
use Laravel\Ai\Files;
use Laravel\Ai\Files\Document;

$stored = Document::fromPath('/path/to/report.pdf')->put();

$response = (new DocumentAnalyzer)->prompt(
    'Summarize this document.',
    attachments: [
        Document::fromId($stored->id),
    ],
);
```

Similarly, stored images may be referenced using the `Image` class:

```php
use Laravel\Ai\Files;
use Laravel\Ai\Files\Image;

$stored = Image::fromPath('/path/to/photo.jpg')->put();

$response = (new ImageAnalyzer)->prompt(
    'What is in this image?',
    attachments: [
        Image::fromId($stored->id),
    ],
);
```

# Vector Stores

Vector stores allow you to create searchable collections of files that can be used for retrieval-augmented generation (RAG). The `Laravel\Ai\Stores` class provides methods for creating, retrieving, and deleting vector stores:

```php
use Laravel\Ai\Stores;

// Create a new vector store...
$store = Stores::create('Knowledge Base');

// Create a store with additional options...
$store = Stores::create(
    name: 'Knowledge Base',
    description: 'Documentation and reference materials.',
    expiresWhenIdleFor: days(30),
);

return $store->id;
```

To retrieve an existing vector store by its ID, use the `get` method:

```php
use Laravel\Ai\Stores;

$store = Stores::get('store_id');

$store->id;
$store->name;
$store->fileCounts;
$store->ready;
```

To delete a vector store, use the `delete` method on the `Stores` class or the store instance:

```php
use Laravel\Ai\Stores;

// Delete by ID...
Stores::delete('store_id');

// Or delete via a store instance...
$store = Stores::get('store_id');

$store->delete();
```

Adding Files to Stores

Once you have a vector store, you may add files to it using the `add` method. Files added to a store are automatically indexed for semantic searching using the file search provider tool:

```php
use Laravel\Ai\Files\Document;
use Laravel\Ai\Stores;

$store = Stores::get('store_id');

// Add a file that has already been stored with the provider...
$document = $store->add('file_id');
$document = $store->add(Document::fromId('file_id'));

// Or, store and add a file in one step...
$document = $store->add(Document::fromPath('/path/to/document.pdf'));
$document = $store->add(Document::fromStorage('manual.pdf'));
$document = $store->add($request->file('document'));

$document->id;
$document->fileId;
```

Typically, when adding previously stored files to vector stores, the returned document ID will match the file's previously assigned ID; however, some vector storage providers may return a new, different "document ID". Therefore, it's recommended that you always store both IDs in your database for future reference.

You may attach metadata to files when adding them to a store. This metadata can later be used to filter search results when using the file search provider tool:

```php
$store->add(Document::fromPath('/path/to/document.pdf'), metadata: [
    'author' => 'Taylor Otwell',
    'department' => 'Engineering',
    'year' => 2026,
]);
```

To remove a file from a store, use the `remove` method:

```php
$store->remove('file_id');
```

Removing a file from a vector store does not remove it from the provider's file storage. To remove a file from the vector store and delete it permanently from file storage, use the `deleteFile` argument:

```php
$store->remove('file_abc123', deleteFile: true);
```

# Failover

When prompting or generating other media, you may provide an array of providers / models to automatically failover to a backup provider / model if a service interruption or rate limit is encountered on the primary provider:

```php
use App\Ai\Agents\SalesCoach;
use Laravel\Ai\Image;

$response = (new SalesCoach)->prompt(
    'Analyze this sales transcript...',
    provider: [Lab::OpenAI, Lab::Anthropic],
);

$image = Image::of('A donut sitting on the kitchen counter')
    ->generate(provider: [Lab::Gemini, Lab::xAI]);
```

# Testing

# Agents

To fake an agent's responses during tests, call the `fake` method on the agent class. You may optionally provide an array of responses or a closure:

```php
use App\Ai\Agents\SalesCoach;
use Laravel\Ai\Prompts\AgentPrompt;

// Automatically generate a fixed response for every prompt...
SalesCoach::fake();

// Provide a list of prompt responses...
SalesCoach::fake([
    'First response',
    'Second response',
]);

// Dynamically handle prompt responses based on the incoming prompt...
SalesCoach::fake(function (AgentPrompt $prompt) {
    return 'Response for: '.$prompt->prompt;
});
```

When `Agent::fake()` is invoked on an agent that returns structured output, Laravel will automatically generate fake data that matches your agent's defined output schema.

After prompting the agent, you may make assertions about the prompts that were received:

```php
use Laravel\Ai\Prompts\AgentPrompt;

SalesCoach::assertPrompted('Analyze this...');

SalesCoach::assertPrompted(function (AgentPrompt $prompt) {
    return $prompt->contains('Analyze');
});

SalesCoach::assertNotPrompted('Missing prompt');

SalesCoach::assertNeverPrompted();
```

For queued agent invocations, use the queued assertion methods:

```php
use Laravel\Ai\QueuedAgentPrompt;
```

```php

SalesCoach::assertQueued('Analyze this...');

SalesCoach::assertQueued(function (QueuedAgentPrompt $prompt) {
    return $prompt->contains('Analyze');
});

SalesCoach::assertNotQueued('Missing prompt');

SalesCoach::assertNeverQueued();
```

To ensure all agent invocations have a corresponding fake response, you may use `preventStrayPrompts`. If an agent is invoked without a defined fake response, an exception will be thrown:

```php
SalesCoach::fake()->preventStrayPrompts();
```

Images

Image generations may be faked by invoking the `fake` method on the `Image` class. Once image has been faked, various assertions may be performed against the recorded image generation prompts:

```php
use Laravel\Ai\Image;
use Laravel\Ai\Prompts\ImagePrompt;
use Laravel\Ai\Prompts\QueuedImagePrompt;

// Automatically generate a fixed response for every prompt...
Image::fake();

// Provide a list of prompt responses...
Image::fake([
    base64_encode($firstImage),
    base64_encode($secondImage),
]);

// Dynamically handle prompt responses based on the incoming prompt...
Image::fake(function (ImagePrompt $prompt) {
    return base64_encode('...');
});
```

After generating images, you may make assertions about the prompts that were received:

```php
Image::assertGenerated(function (ImagePrompt $prompt) {
    return $prompt->contains('sunset') && $prompt->isLandscape();
});

Image::assertNotGenerated('Missing prompt');

Image::assertNothingGenerated();
```

For queued image generations, use the queued assertion methods:

```php
Image::assertQueued(
    fn (QueuedImagePrompt $prompt) => $prompt->contains('sunset')
);

Image::assertNotQueued('Missing prompt');

Image::assertNothingQueued();
```

To ensure all image generations have a corresponding fake response, you may use `preventStrayImages`. If an image is generated without a defined fake response, an exception will be thrown:

```php
Image::fake()->preventStrayImages();
```

# Audio

Audio generations may be faked by invoking the `fake` method on the `Audio` class. Once audio has been faked, various assertions may be performed against the recorded audio generation prompts:

```php
use Laravel\Ai\Audio;
use Laravel\Ai\Prompts\AudioPrompt;
use Laravel\Ai\Prompts\QueuedAudioPrompt;

// Automatically generate a fixed response for every prompt...
Audio::fake();

// Provide a list of prompt responses...
Audio::fake([
    base64_encode($firstAudio),
    base64_encode($secondAudio),
]);
```

```php

// Dynamically handle prompt responses based on the incoming prompt...
Audio::fake(function (AudioPrompt $prompt) {
    return base64_encode('...');
});
```

After generating audio, you may make assertions about the prompts that were received:

```php
Audio::assertGenerated(function (AudioPrompt $prompt) {
    return $prompt->contains('Hello') && $prompt->isFemale();
});

Audio::assertNotGenerated('Missing prompt');

Audio::assertNothingGenerated();
```

For queued audio generations, use the queued assertion methods:

```php
Audio::assertQueued(
    fn (QueuedAudioPrompt $prompt) => $prompt->contains('Hello')
);

Audio::assertNotQueued('Missing prompt');

Audio::assertNothingQueued();
```

To ensure all audio generations have a corresponding fake response, you may use `preventStrayAudio`. If audio is generated without a defined fake response, an exception will be thrown:

```php
Audio::fake()->preventStrayAudio();
```

Transcriptions

Transcription generations may be faked by invoking the `fake` method on the `Transcription` class. Once transcription has been faked, various assertions may be performed against the recorded transcription generation prompts:

```php
use Laravel\Ai\Transcription;
use Laravel\Ai\Prompts\TranscriptionPrompt;
use Laravel\Ai\Prompts\QueuedTranscriptionPrompt;
```

```php

// Automatically generate a fixed response for every prompt...
Transcription::fake();

// Provide a list of prompt responses...
Transcription::fake([
    'First transcription text.',
    'Second transcription text.',
]);

// Dynamically handle prompt responses based on the incoming prompt...
Transcription::fake(function (TranscriptionPrompt $prompt) {
    return 'Transcribed text...';
});
```

After generating transcriptions, you may make assertions about the prompts that were received:

```php
Transcription::assertGenerated(function (TranscriptionPrompt $prompt) {
    return $prompt->language === 'en' && $prompt->isDiarized();
});

Transcription::assertNotGenerated(
    fn (TranscriptionPrompt $prompt) => $prompt->language === 'fr'
);

Transcription::assertNothingGenerated();
```

For queued transcription generations, use the queued assertion methods:

```php
Transcription::assertQueued(
    fn (QueuedTranscriptionPrompt $prompt) => $prompt->isDiarized()
);

Transcription::assertNotQueued(
    fn (QueuedTranscriptionPrompt $prompt) => $prompt->language === 'fr'
);

Transcription::assertNothingQueued();
```

To ensure all transcription generations have a corresponding fake response, you may use `preventStrayTranscriptions`. If a transcription is generated without a defined fake response, an exception will be thrown:

```php
Transcription::fake()->preventStrayTranscriptions();
```

Embeddings

Embeddings generations may be faked by invoking the `fake` method on the `Embeddings` class. Once embeddings has been faked, various assertions may be performed against the recorded embeddings generation prompts:

```php
use Laravel\Ai\Embeddings;
use Laravel\Ai\Prompts\EmbeddingsPrompt;
use Laravel\Ai\Prompts\QueuedEmbeddingsPrompt;

// Automatically generate fake embeddings of the proper dimensions for every
Embeddings::fake();

// Provide a list of prompt responses...
Embeddings::fake([
    [$firstEmbeddingVector],
    [$secondEmbeddingVector],
]);

// Dynamically handle prompt responses based on the incoming prompt...
Embeddings::fake(function (EmbeddingsPrompt $prompt) {
    return array_map(
        fn () => Embeddings::fakeEmbedding($prompt->dimensions),
        $prompt->inputs
    );
});
```

After generating embeddings, you may make assertions about the prompts that were received:

```php
Embeddings::assertGenerated(function (EmbeddingsPrompt $prompt) {
    return $prompt->contains('Laravel') && $prompt->dimensions === 1536;
});

Embeddings::assertNotGenerated(
    fn (EmbeddingsPrompt $prompt) => $prompt->contains('Other')
);

Embeddings::assertNothingGenerated();
```

For queued embeddings generations, use the queued assertion methods:

```php
Embeddings::assertQueued(
    fn (QueuedEmbeddingsPrompt $prompt) => $prompt->contains('Laravel')
);

Embeddings::assertNotQueued(
    fn (QueuedEmbeddingsPrompt $prompt) => $prompt->contains('Other')
);

Embeddings::assertNothingQueued();
```

To ensure all embeddings generations have a corresponding fake response, you may use `preventStrayEmbeddings`. If embeddings are generated without a defined fake response, an exception will be thrown:

```php
Embeddings::fake()->preventStrayEmbeddings();
```

# Reranking

Reranking operations may be faked by invoking the `fake` method on the `Reranking` class:

```php
use Laravel\Ai\Reranking;
use Laravel\Ai\Prompts\RerankingPrompt;
use Laravel\Ai\Responses\Data\RankedDocument;

// Automatically generate a fake reranked responses...
Reranking::fake();

// Provide custom responses...
Reranking::fake([
    [
        new RankedDocument(index: 0, document: 'First', score: 0.95),
        new RankedDocument(index: 1, document: 'Second', score: 0.80),
    ],
]);
```

After reranking, you may make assertions about the operations that were performed:

```php
Reranking::assertReranked(function (RerankingPrompt $prompt) {
    return $prompt->contains('Laravel') && $prompt->limit === 5;
});

Reranking::assertNotReranked(
    fn (RerankingPrompt $prompt) => $prompt->contains('Django')
);

Reranking::assertNothingReranked();
```

Files

File operations may be faked by invoking the `fake` method on the `Files` class:

```php
use Laravel\Ai\Files;

Files::fake();
```

Once file operations have been faked, you may make assertions about the uploads and deletions that occurred:

```php
use Laravel\Ai\Contracts\Files\StorableFile;
use Laravel\Ai\Files\Document;

// Store files...
Document::fromString('Hello, Laravel!', mime: 'text/plain')
    ->as('hello.txt')
    ->put();

// Make assertions...
Files::assertStored(fn (StorableFile $file) =>
    (string) $file === 'Hello, Laravel!' &&
        $file->mimeType() === 'text/plain';
);

Files::assertNotStored(fn (StorableFile $file) =>
    (string) $file === 'Hello, World!'
);

Files::assertNothingStored();
```

For asserting against file deletions, you may pass a file ID:

```php
Files::assertDeleted('file-id');
Files::assertNotDeleted('file-id');
Files::assertNothingDeleted();
```

Vector Stores

Vector store operations may be faked by invoking the `fake` method on the `Stores` class. Faking stores will also fake file operations automatically:

```php
use Laravel\Ai\Stores;

Stores::fake();
```

Once store operations have been faked, you may make assertions about the stores that were created or deleted:

```php
use Laravel\Ai\Stores;

// Create store ...
$store = Stores::create('Knowledge Base');

// Make assertions ...
Stores::assertCreated('Knowledge Base');

Stores::assertCreated(fn (string $name, ?string $description) =>
    $name === 'Knowledge Base'
);

Stores::assertNotCreated('Other Store');

Stores::assertNothingCreated();
```

For asserting against store deletions, you may provide the store ID:

```php
Stores::assertDeleted('store_id');
Stores::assertNotDeleted('other_store_id');
Stores::assertNothingDeleted();
```

To assert files were added or removed from a store, use the assertion methods on a given `Store` instance:

```php
Stores::fake();

$store = Stores::get('store_id');

// Add / remove files ...
$store->add('added_id');
```

```php
$store->remove('removed_id');

// Make assertions...
$store->assertAdded('added_id');
$store->assertRemoved('removed_id');

$store->assertNotAdded('other_file_id');
$store->assertNotRemoved('other_file_id');
```

If a file is stored in the provider's file storage and added to a vector store in the same request, you may not know the file's provider ID. In this case, you can pass a closure to the `assertAdded` method to assert against the content of the added file:

```php
use Laravel\Ai\Contracts\Files\StorableFile;
use Laravel\Ai\Files\Document;

$store->add(Document::fromString('Hello, World!', 'text/plain')->as('hello.tx

$store->assertAdded(fn (StorableFile $file) => $file->name() === 'hello.txt')
$store->assertAdded(fn (StorableFile $file) => $file->content() === 'Hello, W
```

# Events

The Laravel AI SDK dispatches a variety of events, including:

- `AddingFileToStore`
- `AgentPrompted`
- `AgentStreamed`
- `AudioGenerated`
- `CreatingStore`
- `EmbeddingsGenerated`
- `FileAddedToStore`
- `FileDeleted`
- `FileRemovedFromStore`
- `FileStored`
- `GeneratingAudio`
- `GeneratingEmbeddings`

`GeneratingImage`

`GeneratingTranscription`

`ImageGenerated`

`InvokingTool`

`PromptingAgent`

`RemovingFileFromStore`

`Reranked`

`Reranking`

`StoreCreated`

`StoringFile`

`StreamingAgent`

`ToolInvoked`

`TranscriptionGenerated`

You can listen to any of these events to log or store AI SDK usage information.

Pint

Sail

Sanctum

Socialite

Telescope

Pulse

Reverb

Echo

Resources

Documentation

Starter Kits

Release Notes

Blog

News

Community

Larabelles

Learn

Jobs

Careers

Trust

Partners

Curotec

Active Logic

Tighten

Kirschbaum

Vehikl

byte5

Redberry

Jump24

More Partners

Laravel